

# **Interfacing SD/MMC Cards With TMS320F28xxx DSCs**

*Pradeep Shinde, Tim Love*

## **ABSTRACT**

This application report and associated code implements the interfacing of Secure Digital (SD) or Multi-Media Card (MMC) types of Flash memory cards with the TMS320F28xxx digital signal controllers (DSCs). These cards can use serial peripheral interface (SPI) or parallel (SD) interface to connect to the host. This application report demonstrates the SPI mode and includes the schematics showing the interconnections and the software routines. This is basic driver software, providing basic functionality. You can add the SD/MMC commands and additional functions required per application. This document covers SD versions 1.10 and the new 2.00 specifications, meant for SD high capacity (SDHC) cards.

Project collateral and source code discussed in this application report can be downloaded from <http://www.ti.com/lit/zip/SPRAAO7>.

## **Contents**

1	Introduction .....	2
2	SD/MMC Overview .....	3
3	Hardware Description .....	3
4	Software Description.....	4
5	SD Specifications and Compatibility .....	11
6	Conclusion .....	12
7	References .....	12
Appendix A	Schematic .....	13
Appendix B	Source File and Function Listing.....	14

## **List of Figures**

1	Daughter Board Interface to F2808 eZdsp .....	2
2	Interconnections Between TMS320F28xxx and SD Card – SPI Mode.....	4
3	28xxx_SD_SPI Directory Structure.....	5
4	Software Flow .....	6
5	Breakpoint View.....	9
6	Register Contents in Watch Window.....	9
7	View Memory .....	10
8	Write Data in Read_buffer .....	10
9	Erased Data in Read_buffer .....	11
A-1	Dual SD Card Interface Schematic .....	13

## **List of Tables**

B-1	Source File and Function Listing .....	14
-----	--	----

## 1 Introduction

Due to their peripheral rich architecture, TMS320F28xxx DSCs are popular for mid-size applications. Many of these applications require a removable, small size, high-density storage media (data logging, audio, wireless/RF, boot load via SPI, etc.). Flash memory cards are suitable for this purpose. SD and MMC stand out among various brands of such cards in the market today due to their features of content protection for recordable media (CPRM), networking capability and small size. The secure digital input output (SDIO) card is an interface that extends the functionality of a system with SD card slots. SDIO cards such as camera, Bluetooth, FM radio, GPS, voice recorder, digital TV tuner, and 802.11b/g are already available. SD and MMC cards have similar specifications. The SD Memory card protocol is designed to be a superset of MMC's version 2.11 protocol. The SD card is used for this application report. It is assumed that you have a general knowledge of SD, SDIO and MMC cards. You can accommodate for the MMC on an application-specific basis. For more detailed information regarding complete specifications and the differences between the SD and MMC cards, see the websites of the respective standards' associations indicated below:

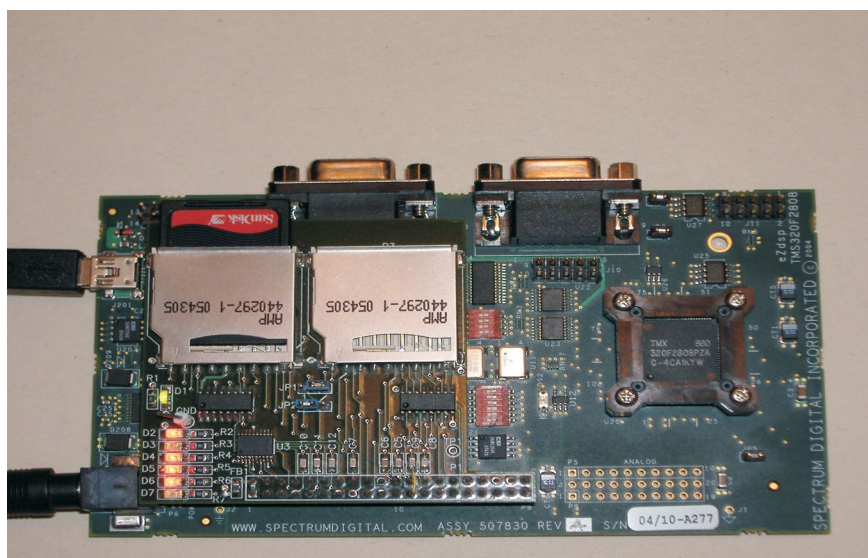
- <http://www.sdcard.org/>
- <http://www.mmca.org/home>

A daughter board for the TMS320F2808 eZdsp board was designed to develop this interface. This card is configurable for the SPI as well as the SD mode and for up to two SD cards. The code being tested using the SD cards from the older 1.10 version and the newer 2.00 version is from a few different vendors. A complete reference design is shown, including a schematic and the software to accomplish basic interface required to add the SD Card to your project. The daughter board connection to the F2808 eZdsp is shown in [Figure 1](#). The associated source code serves as the driver software. It provides the routines to establish communication, card initialization and data read, write and erase functions. You can add any additional higher-level software as required for the application. Note that the SD Card uses FAT16/32 file format for organizing files. The file allocation table (FAT) is the format that helps arrange different data files on a storage media. This interface is also applicable to the F281x DSCs. The hardware interface is the same but the software would require minor changes.

---

**Note:** FAT software is out of scope for this application report.

---



**Figure 1. Daughter Board Interface to F2808 eZdsp**

## 2 SD/MMC Overview

As of April 2007, SD cards up to eight GB capacity are available in the market and the specifications are drawn for the cards up to 32 GB (high capacity cards). They come with a 9-pin edge connector and connect to the host controller via SPI or 4-bit parallel (SD) interface. Extra pins add provisions to detect the presence of the card in the adapter and the write protection. The interface mode is selected at power up and can not be changed until the card is powered down. The version 1.10 specifications support up to a 12.5 MB/sec interface speed with full clock range of 0 MHz-25 MHz and the operation for a supply voltage range of 2.0 V – 3.6 V. New SDLV cards can operate in the voltage range of 1.6 V – 3.6 V.

Communication between the host and the SD card is controlled by the host controller (master) and takes place in a question/answer type token passing. The host sends *Commands* to all cards (broadcast) on the network or to a single card (addressed) and the card returns with a *Response*. Each SPI and SD type of protocol defines the formats for commands and responses. These are found in the SD Physical Layer Specifications [4].

An SD Card uses FAT file system for storing files. Cards up to 2-GB capacity use the FAT16 file format and those greater than 2 GB use FAT32.

The SD Card Association (SDA) released version 2.00 of the specification in September 2006. This new SDA specification enables the SD cards to reach higher capacities: 4 GB to 32 GB. The data transfer speed is increased up to 25 MB/sec; as clock speed goes up to 50 MHz.

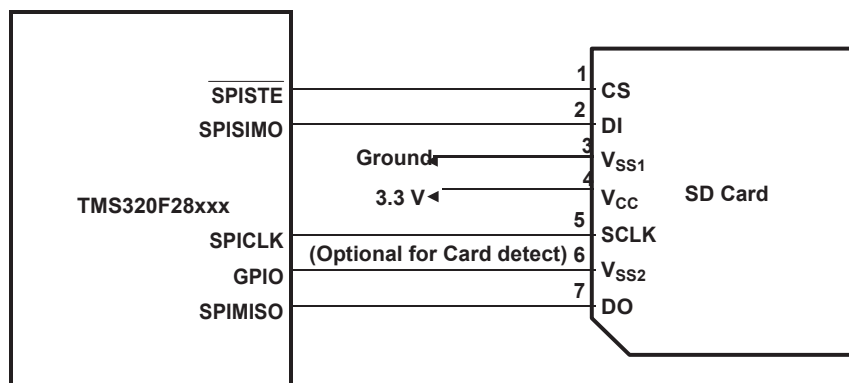
## 3 Hardware Description

The TMS320F28xxx DSCs support an onboard SPI peripheral. This should be the preferred interface for applications using cards with smaller capacities, up to 2 GB. For higher capacity cards, you may want a higher data transfer speed and choose the SD mode. The SD interface is implemented using general-purpose input/output (GPIO) pins to meet particular SD specifications.

### 3.1 Generic Interface

The interconnections for SPI modes are shown in [Figure 2](#). The pullup resistors are specified as 10 K – 100 K for all the input signals (as opposed to a floating condition when card drivers are in high-impedance mode). CMD and DATA pins need to be pulled high. Though SPI mode does not use all DATA pins, the host must pull them high. The write protection (WP) pin is also pulled high. Filter capacitors are advised on each signal; however, the total load capacitance ( $C_{\text{HOST}} + C_{\text{BUS}} + N \cdot C_{\text{CARD}}$ ) should not exceed 100 pF if seven cards are operating at a clock speed up to 20 MHz. A standard SD card can work with a supply voltage of 2.0 (min) to 3.6 (max) and draw up to 200 mA current; therefore, the F28xxx devices  $V_{\text{DDIO}}$  voltage rail of 3.3 V is used to power the card. For interfacing to SD cards requiring a supply voltage other than 3.3 V, a voltage level translation device like the Texas Instruments SN74AVCA406 should be used. Electro static discharge (ESD) protection can be added as additional precaution. The SD card specifies Human Body Model of  $\pm 4$  KV, 100 pf/1.5  $\Omega$  as ESD parameters.

Most sockets provide an additional switch for card detect and write protect. There are two ways to achieve card detection. A ground line ( $V_{SS}$ ) can be used with a pullup resistor to read the switch closer. Alternately, a return Ground connection ( $V_{SS2}$ ) can be read on pin 6 in [Figure 2](#).



**Figure 2. Interconnections Between TMS320F28xxx and SD Card – SPI Mode**

### 3.2 Test Circuit

The schematic of the daughter board used to develop the code is shown in the [Appendix A](#). Jumpers JP1 and JP2 configure it for either SPI (pins 1 and 2 shorted) or SD (pins 2 and 3 shorted) mode. Additional hardware includes ESD protection for each signal by using TPD6E001 (U1, U2, and U4). LEDs indicate different activities while running the associated code. The second SD card adapter (P2 and P3) was added to test the network capability.

---

**Note:** The network capability is out of scope for this application report.

---

## 4 Software Description

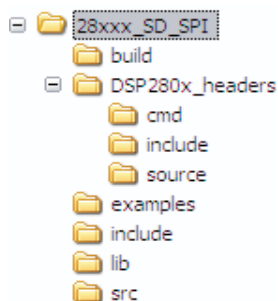
The software provided with this application report includes a library and an example project demonstrating how to use this library. The project used to build the library is also provided to enable modification as needed per application.

The library functions as a device driver for communication between the TMS320F28xxx devices and the SD card using SPI communication. The library provides these basic interface functions:

- Card insertion detection
- Power up/card initialization
- Register manipulation
- Write, read, and erase data

Write protection, error checking, card lock/unlock, copyright protection commands, and switch function are out of scope for this device driver. This functionality can be added on an application-specific basis.

The software is self contained and extracts with the 28xxx\_SD\_SPI folder as the base directory. This code uses several files from the *C280x, C2801x C/C++ Header Files and Peripheral Examples* ([SPRC191](#)) [5]. [Figure 3](#) shows the directory structure of the F28xxx\_SD\_SPI directory.



**Figure 3. 28xxx\_SD\_SPI Directory Structure**

The build folder contains the library project file. The DSP280x\_headers directory contains all files used from the *C280x, C2801x C/C++ Header Files and Peripheral Examples* ([SPRC191](#)) [5]. The examples directory contains the example project. The *include* folder contains header files used by the library project. The lib folder contains the device driver library. The src folder contains all source files for the library. The code was tested using Code Composer Studio™ software version 3.1 using the TMS320F28xxx Code Generation Tools version 4.1.3.

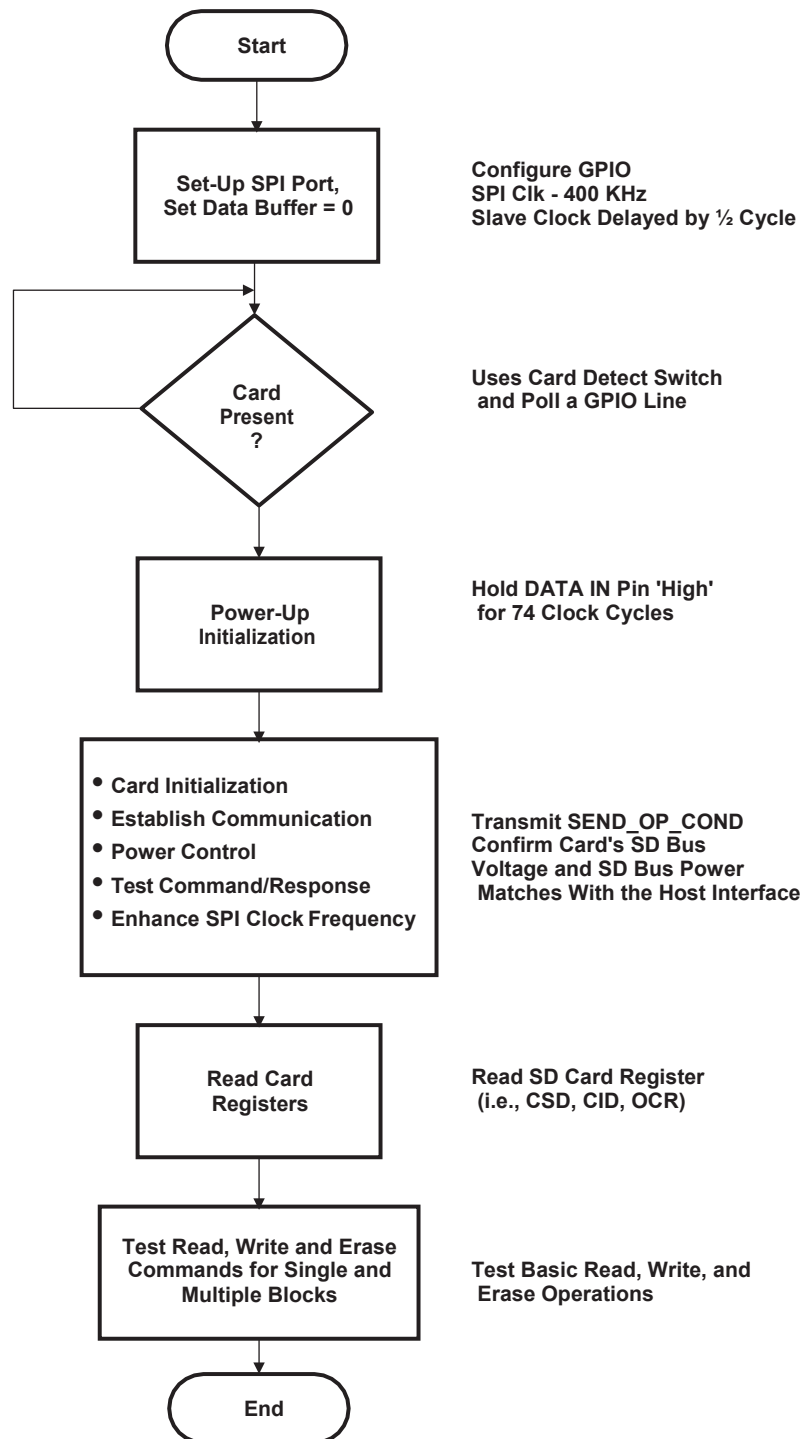
## 4.1 Software Flow

[Figure 4](#) describes the software flow used in the example. This software flow should be followed by most applications, especially the initialization portion. There is a fixed sequence in which the card should be reset and initialized. For more information on the details of this procedure, see the SD Group's specifications [4]. Once initialization is finished, the host can read the card registers and read, write, and erase data on the card. The example project provided contains the function calls to demonstrate the software flow.

### 4.1.1 Initialization

Since the F28xxx device is communicating to the SD card through the SPI, the first step of the process is to configure the SPI of the device. This is performed through the `spi_initialization()` function. This is located in the `SD_SPI_Initialization.c` file. For a complete function listing by source file, please see [Appendix B](#).

The SD daughter board interfaces to SPI-A of the F2808 device. This software uses 4-wire SPI, meaning a single chip select (CS) signal is used to select one SD adapter. The `SPISTE(CS)` of the F28xxx is configured as a GPIO output and pulled high and low manually to meet timing specifications of the SD card. For the designs supporting 2+ adapters, 3-wire SPI protocol and GPIO pins for CS signals, the device must be used in this manner. Communication is discussed later in [Section 4.1.2](#).


**Figure 4. Software Flow**

After SPI initialization is finished, card insertion is checked. The  $V_{SS2}$  line from the SD card interface is driven low when a card is inserted. This signal is connected to GPIO34 of the F2808 eZdsp. The `check_card_insertion()` function continually monitors this GPIO signal to detect when it is driven low; this is written as a continuous loop. For application purposes, a timeout can be added to check for card insertion only for a certain amount of time or the  $V_{SS2}$  signal can be interfaced to a GPIO that can be used as an external interrupt.

Once the card has been detected, the SPI transmits 0xFF00 10 times to cause the DATA IN signal of the SD card to be high for the 74 cycles required for power up initialization. The reason the data transmitted from the SPI is left aligned is because the SPI transmits MSB first per protocol [3] and the SPI is configured for eight bit transmission.

Once a card has passed the power up initialization, it needs to go through the card initialization; the `sd_initialization()` function performs this function. The first phase of card initialization is to transmit the `GO_IDLE_STATE` command with the CS line pulled low to transfer to the SPI mode. A command consists of six bytes: the first byte is the command, the next four bytes are for status, address, or data and the last byte is for cyclic redundancy code (CRC) check. In the SPI mode, the CRC is disabled by default so this example just sends a dummy CRC value (0xFF00) for most commands. The `GO_IDLE_STATE` command must have a valid CRC value at the time it is transmitted because the card is in the SDIO mode. This CRC value for this command is specified in the SD Physical Layer Specifications [4].

Once the `GO_IDLE_STATE` command has passed successfully, the `SEND_IF_COND` command must be sent to determine the version of the card. The `sd_version1_initialization()` or `sd_version2_initialization()` is called depending on the response from the `SEND_IF_COND`. Both initialization functions determine the operating voltage of the device. The `sd_version2_initialization()` also verifies that the inserted card is an SDHC. Once these functions have finished, the inserted card is initialized. At this point, the SPI clock can be increased. Before this point, the SPI clock must be configured for a range of 100 KHz to 400 KHz for initialization per SD protocol [4].

*Optional:* Also included in the initialization portion of the software is the `led_initialization()` used for status LEDs on the daughter board. This is completely optional and based on the application.

#### 4.1.2 Communication

SPI communication to the SD card is achieved from the `spi_xmit_byte()` function. This function accepts a data byte as its parameter and places this byte in the `SPITXBUF` to transmit.

The `spi_xmit_command()` function is used to transmit all commands to the SD card. This function accepts the command, data, and CRC that is transmitted to the card. The function checks that the sent command is for data manipulation. If the command has data associated to an address, the data is multiplied by 512 for the sector size of the data. This data is later placed in the command as the proper address. The CRC is also checked; if the CRC is needed, the `sd_crc7` function is called for the 6th byte of the command. As mentioned earlier, CRC verification is disabled by default for the SPI mode except for two specific commands; the example only uses the CRC for these commands. CRC verification can be enabled for specific applications and then this function can perform the CRC.

The `sd_command_response()` function checks for specific responses from commands. The specific commands checked are the `IN_IDLE_STATE` and `SUCCESS` responses. If these responses are received, the function execution halts. If neither of these responses is returned, the function checks if the `DATA_OUT` signal is still high. If it is not high, then an error has occurred and `sd_error()` is called. Once again, error checking is out of the scope of this project. This needs to be written for each specific application. If a command needs a response other than the `IN_IDLE_STATE` or `SUCCESS`, this is checked for in the specific function waiting for the response. The card response is stored in a variable simply named response. There is also a `RESET_RESPONSE` command. This command sets the response to 0xFF00. This is called numerous times throughout the code to reset the response before checking for a specific value.

The `EIGHT_CLOCK_CYCLE_DELAY` command is also called numerous times throughout the code. This command transmits 0xFF00 in order for the SPI clock to be active for eight clock cycles. This is required per SD protocol after receiving a response [4].

### 4.1.3 Register Manipulation

Reading certain registers is applicable to most applications. In this example, it is not needed but it is included for completeness. The software provides calls for reading the CSD, CID, and OCR registers. For more detailed information on these registers, see the *SD Physical Layer Specifications* [4]. These registers are read from calling the `sd_read_register()` function. The contents of these registers are saved to the `csd_contents`, `cid_contents`, and `ocr_contents` arrays, respectively, in the `sd_ocr_response()` and `sd_cid_csd_response()` functions. The contents of these registers can be viewed by adding the above arrays to the Code Composer Studio Watch Window during debug.

### 4.1.4 Read, Write, and Erase

The final functionality provided in the software is the read, write, and erase for single and multiple blocks of data. The single block write, read, and erase is tested first in the example. The write block is accomplished using the `sd_write_block()` function. This function requires the sector address and a pointer to an array of data to be written to the card. In case of the SDHC card, the SD card writes according to the sector number as opposed to the address as specified in the *SD Physical Layer Specification* [4]. The example uses an array named `write_buffer`, initialized continuously from `0x0000` to `0xFF00`, to test functionality. The `sd_write_block` function transmits the `WRITE_BLOCK` command, waits for response, and then transmits the data in the buffer byte by byte. The host must then transmit the `SEND_STATUS` command to check if all data was written correctly. The `sd_write_multiple_block()` function has the same functionality as the `sd_write_block()` function except that it writes to multiple blocks.

The `sd_read_block()` function is the functionally the same as the `sd_write_block()` function. The sector address and the array to store the data read from the card is required by the function. After the `READ_SINGLE_BLOCK` command is transmitted, the host waits for a response, then reads the data and stores it byte by byte to the array being pointed to. The example uses an array named `read_buffer` to store data. The `sd_read_multiple_block()` function has the same functionality as the `sd_read_block()` function except that it reads multiple blocks.

The `sd_erase_block()` function works for both single block and multiple block erase. The function accepts the starting sector and the number of sectors to erase. The function transmits the first sector to be erased, then transmits the total numbers of sectors, and then transmits the `ERASE` command. The function ends by continuously monitoring the `DATA OUT` line from the `CARD`. While erasing, this line is constantly low; once the card is finished erasing, this line is released high once again.

Note that the Erase operation deletes the data from the Card. If the contents of the Card are old, it will be lost.

## 4.2 Testing Example

To properly test the example, the watch window, memory window, and breakpoints need to be utilized within the Code Composer Studio software. The following steps were followed to test the SD interface daughter board with the F2808 eZdsp and the Code Composer Studio:

1. Connect the daughter board with the SD card inserted in slot P2 to the F2808 eZdsp and power the board as shown in [Figure 1](#).
2. Start Code Composer Studio with the F2808 eZdsp emulation driver selected in the Code Composer Studio setup utility.
3. Open, build, and load `Example_28xxx_SD_SPI.pjt` by selecting `Project` → `Open` followed by `Project` → `Rebuild All` and finally `File` → `Load Program`. As supplied, the project is tested from RAM. This can easily be ported to flash by modifying the linker command file (`.cmd`).
4. Enter the `main()` function by selecting `Debug` → `Go Main`, once the project is loaded.



- Set breakpoints in order to properly view the data and register contents in the watch window and memory window. This is illustrated in [Figure 5](#).

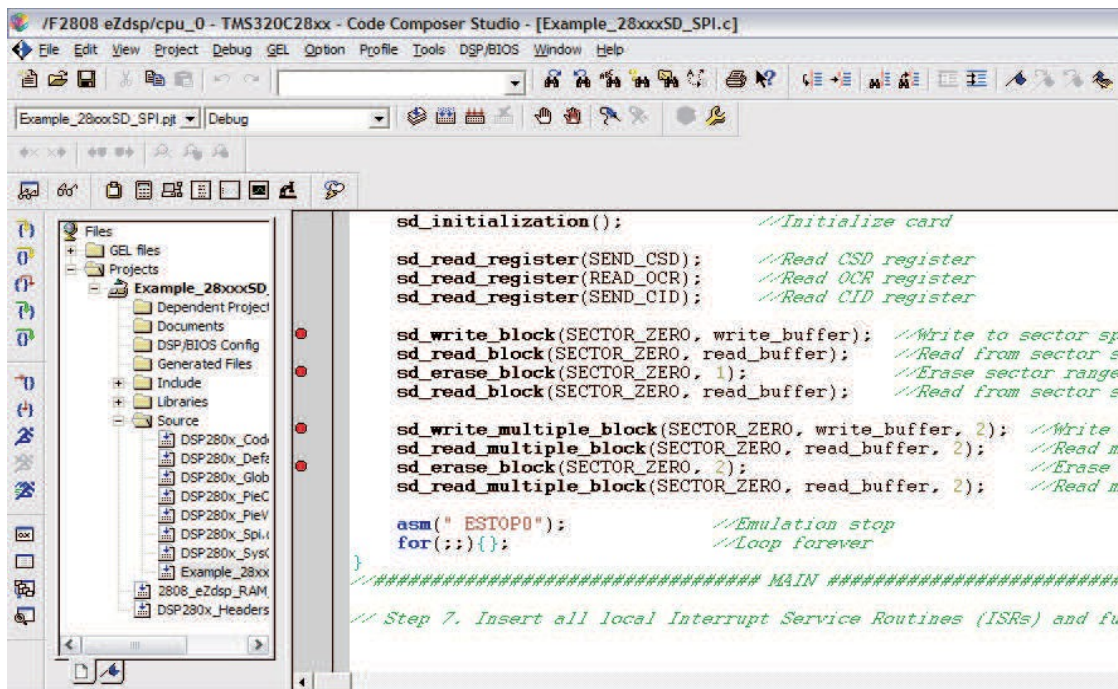


Figure 5. Breakpoint View

- Run to the first breakpoint by selecting Debug → Run. Once the first breakpoint is met, the card is initialized and the registers are read. The watch window is opened by selecting View → Watch Window. The contents of the registers can be viewed in the watch window by typing the array names `csd_contents`, `ocr_contents`, and `cid_contents` in the Watch 1 tab. This is illustrated in [Figure 6](#).

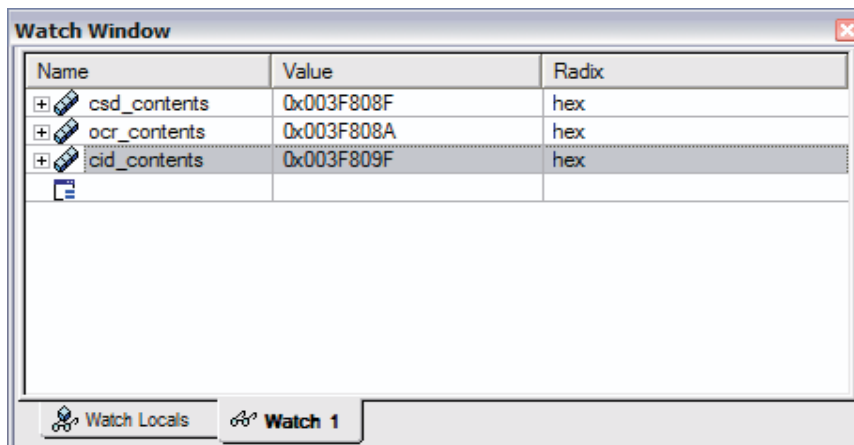
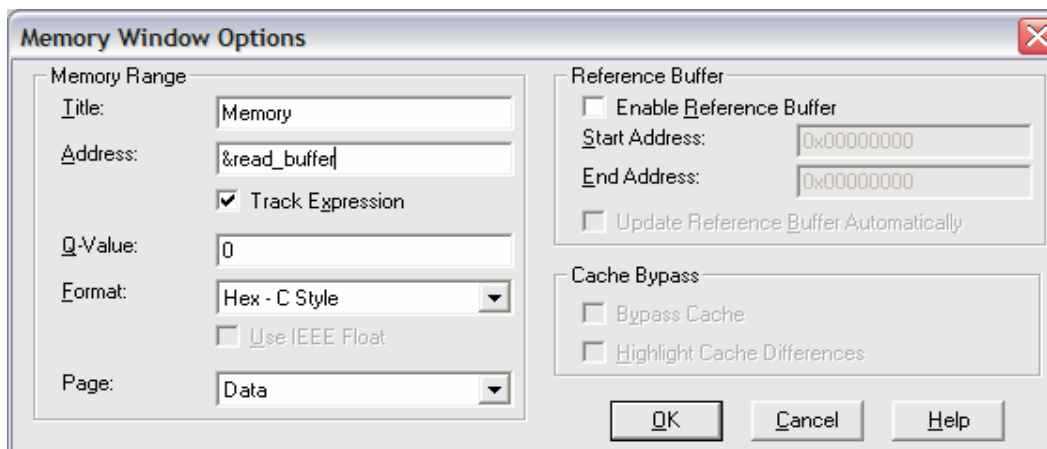


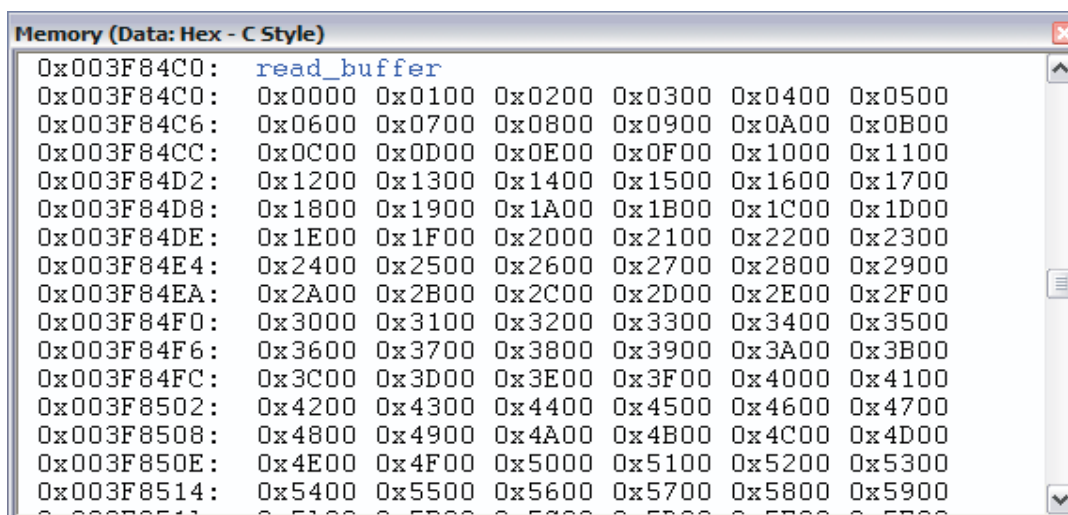
Figure 6. Register Contents in Watch Window

- Once the registers are read, the data is written, read, and erased on the card. After the data is written to the card, the card memory is read and this data is stored in the read\_buffer. This buffer can be examined by selecting View → Memory and specifying &read\_buffer at the address shown in Figure 7.



**Figure 7. View Memory**

For this example, the data written to the card should be 0x0000 to 0xFF00, continuously, as shown in Figure 8.



**Figure 8. Write Data in Read\_buffer**

The read after an erase should show the card memory as either 0x0000 or 0xFF00 depending on the card manufacturer. This is illustrated in [Figure 9](#).

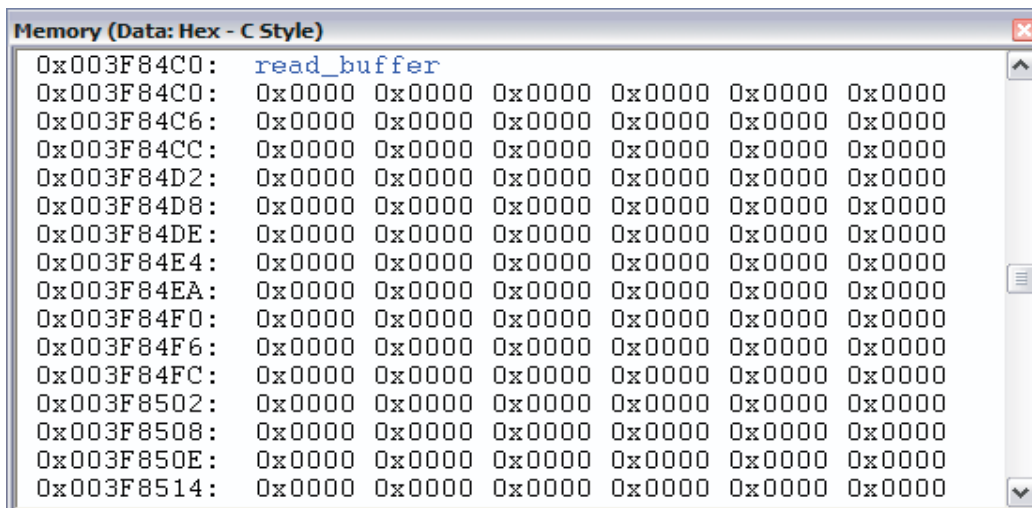


Figure 9. Erased Data in Read\_buffer

### 4.3 Application Integration

Applications using this driver need to include the 28xxx\_SD\_SPI.lib and SD.h file (provided in the lib and included directories) in their project. The 28xxx\_SD\_SPI.lib is generated from the 28xxx\_SD\_SPI\_lib.pjt located in the build directory. This library is composed of the following source files:

- SD\_SPI\_Erase.c
- SD\_SPI\_Initialization.c
- SD\_SPI\_Read.c
- SD\_SPI\_Registers.c
- SD\_SPI\_Transmission.c
- SD\_SPI\_Write.c

For any modification to the library, open 28xxx\_SD\_SPI\_lib.pjt, make the modifications, and rebuild the project. The new library file will be created in the library folder of the directory structure shown in [Figure 3](#). A complete function listing for this library by source file is given in [Appendix B](#).

## 5 SD Specifications and Compatibility

The SDA is continuously working on enhancing the physical layer specifications as the card capacity is increasing. The code was initially designed using version 1.10 specifications. Version 1.10 starts by sending CMD0, which is a software reset, and results in the card initialization. It then enters in the SPI mode, if CS is pulled low at this stage. Version 2.00 of the SD Card Physical Layer Specifications describes a modified Card initialization and identification flow. After powering, it also starts by sending CMD0; however, it is then mandatory to send CMD8 for the version 2.00 cards. CMD8 is used to verify SD Memory Card operating conditions.

Cards with both of these specifications are available in the market. Those having less than 2 GB capacity normally follow version 1.10 or earlier specifications. SDHC are the newer SD memory cards, based on the new SDA 2.00 specifications. Since, SDHC memory accessing works differently than standard SD cards, the host software must account for this. The associated driver software accounts for this, thus it supports SDHC (version 2.00) cards as well as older (version 1.10).

## 6 Conclusion

This application report shows that the F28xxx devices are capable of interfacing to an SD/MMC card through the SPI, allowing you to supply one interface for data acquisition and enabling functionality to another SD interface. This document provides designers with a ready-made device driver and reference design to accelerate and simplify design.

## 7 References

1. *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Digital Signal Processors* ([SPRS174](#))
2. *TMS320F2809, TMS320F2808, TMS320F2806, TMS320F2802, TMS320F2801, TMS320C2802, TMS320C2801, and TMS320F2801x DSPs* ([SPRS230](#))
3. *TMS320x28x, 28xxx Serial Peripheral Interface (SPI) Reference Guide* ([SPRU059](#))
4. *Physical Layer Simplified Specifications, SD Specifications, SD Card Association Technical Committee – Part 1*, (Version 1.10 October 2004 and Version 2.00 September, 2006)
5. *C280x, C2801x C/C++ Header Files and Peripheral Examples* ([SPRC191](#))
6. *TMS320F28044 Digital Signal Processor* ([SPRS357](#))
7. *TMS320F28332, TMS320F28334, TMS320F28335 Digital Signal Controllers (DSCs)* ([SPRS439](#))
8. *TMS320R2811, TMS320R2812 Digital Signal Processors* ([SPRS257](#))



## Appendix B Source File and Function Listing

### B.1 Source File and Function Listing

Table B-1 shows a complete function listing by source file.

**Table B-1. Source File and Function Listing**

Files	Functions	Description
SD_SPI_Erase.c	sd_erase_block()	Performs erase procedure
SD_SPI_Initialization.c	spi_initialization() led_initialization() sd_card_insertion() sd_initialization() sd_version1_initialization() sd_version2_initialization()	Initializes SPI-A of F28xxx Initializes LEDs of daughter board Checks for card insertion Begins card initialization Completes V1.10 initialization Completes V2.00 initialization
SD_SPI_Read.c	sd_read_block() sd_read_multiple_block() sd_data_response()	Initiates single block read Initiates multiple block read Performs read operation
SD_SPI_Registers.c	sd_read_register() sd_ocr_response() sd_cid_csd_response() sd_send_status()	Initiates register read operation Reads OCR Register contents Reads CID and CSD Register contents Verifies card status from status register
SD_SPI_Transmission.c	spi_xmit_byte() spi_xmit_command() sd_crc7() sd_command_response() sd_error()	Transmits byte through SPI-A Transmits full command Performs CRC7 operation Receives command response Trap function for errors
SD_SPI_Write.c	sd_write_block() sd_write_multiple_block() sd_write_data()	Initiates single block write Initiates multiple block write Performs write operation
SD.h		Contains all commands, data tokens, global prototypes, and global variables

## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale ([www.ti.com/legal/termsofsale.html](http://www.ti.com/legal/termsofsale.html)) or other applicable terms available either on [ti.com](http://ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2019, Texas Instruments Incorporated